JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2005

Vol. 6, No. 2, March–April 2006

# Implementing Object-Z with *Perfect Developer*

**Brian Stevens**, Department of Information Systems, Defence College of Management and Technology, Cranfield University, Shrivenham, Swindon. SN6 8LA. UK

Object oriented development is the methodology of choice for a wide range of applications but those developing critical systems have stayed with techniques with a mathematical basis. Object-Z is a formal specification language that has attempted to bring the benefits of an object oriented approach to critical systems. However, it lacks an obvious route to implementation. This paper describes how Object-Z may be implemented using *Perfect Developer*, an OO language that supports verification and validation.

## 1 INTRODUCTION

Object oriented (OO) software design has become the methodology of choice for a large number of software engineers and along with this the graphical notation of the Unified Modelling Language (UML)[1] has become the *de facto* industry standard for OO software design. Unfortunately these graphical notations are not suitable for capturing the potentially complex and subtle restrictions that apply to critical systems and to help resolve this the Object Constraint Language(OCL)[2] has been incorporated into the UML standard. OCL allows for the precise specification of a range of invariants and pre and post conditions on object methods.

Whilst commercial software development has adopted the OO paradigm whole heartedly, those developing critical systems have prefered to stay with proven methodologies and languages and have refined the mathematics behind proving software. One attempt at embracing both worlds is the Object-Z specification language[3]. Object-Z is a formal specification language that was developed at the University of Queensland and is a derivative of the state based formal specification language Z[4]. Although Object-Z provides a mechanism for the precise and unambigous specification of an OO software model it is lacking the tool support to enable implementation whilst maintaining verification of the refinement steps. Some attempts have been made to formalise the implementation of Object-Z models in Eiffel via the BON methodology[5]. This paper takes a different approach and demonstrates how an Object-Z specification can be expressed in *Perfect*[6] and refined towards an implementation whilst maintaining the ability to prove the specification and implementation formally.

## 2 BACKGROUND

### Object-Z

Object-Z was developed in the late 1980s as a collaboration between the Software Validation and Research Centre (University of Queensland, Australia) and the Overseas Telecommunications Corporation (OTC) of Australia as an enhancement of the Z language by adding structure to it. Object-Z provides OO constructs such as classes, attributes, methods and inheritance etc. whilst maintaining formal semantics. Tool support is provided via a type checker and graphical editor. The well used *Credit Card* example from [7] is illustrated in Figure 1.

### *Perfect Developer*

*Perfect Developer* is a formal specification and implementation tool aimed at software development and software engineers without a strong mathematical background. It provides a notation for state based specification that can be directly input from a conventional keyboard and makes use of plain english (e.g. $\forall$ is *forall*). *Perfect Developer* has been designed to provide

- object oriented design;
- automated reasoning;
- automatic code generation.

It extends the Eiffel concept of *Design By Contract* to one of *Verified Design By Contract* and provides C++ or Java code ready to compile.

The *Perfect* language is described in [6] and the basic structure of a *Perfect* class is illustrated in Figure 2.

Following the preamble, a class declaration comprises one or more of the following sections:

#### abstract

- *var*iable declarations;
  These describe the abstract data model and for the *CreditCard* example would include

  ```
  var
     limit : nat,
     balance : int;
  ```

$CreditCard$

$\upharpoonright(limit, balance, INIT, withdraw, deposit, withdrawAvail)$

$limit : \mathbb{N}$

$limit \in 1000, 2000, 5000$

$balance : \mathbb{Z}$

$balance + limit \geqslant 0$

$INIT$

$balance = 0$

$withdraw$

$\Delta(balance)$
$amount? : \mathbb{N}$

$amount? \leqslant balance + limit$
$balance' = balance - amount?$

$deposit$

$\Delta(balance)$
$amount? : \mathbb{N}$

$balance' = balance + amount?$

$withdrawAvail$

$\Delta(balance)$
$amount! : \mathbb{N}$

$amount! = balance + limit$
$balance' = -limit$

Figure 1: Credit Card

- *invariant* declarations;
  These describe any constraints that apply to the abstract variables and for the *Creditcard* example would include

```
invariant
   limit in set of nat {1000, 2000, 5000},
   balance + limit >= 0 ;
```

- Abstract method declarations;
  These describe methods to obtain derived data that may be treated as variables (*cf* secondary variables in Object-Z). Although there are no secondary

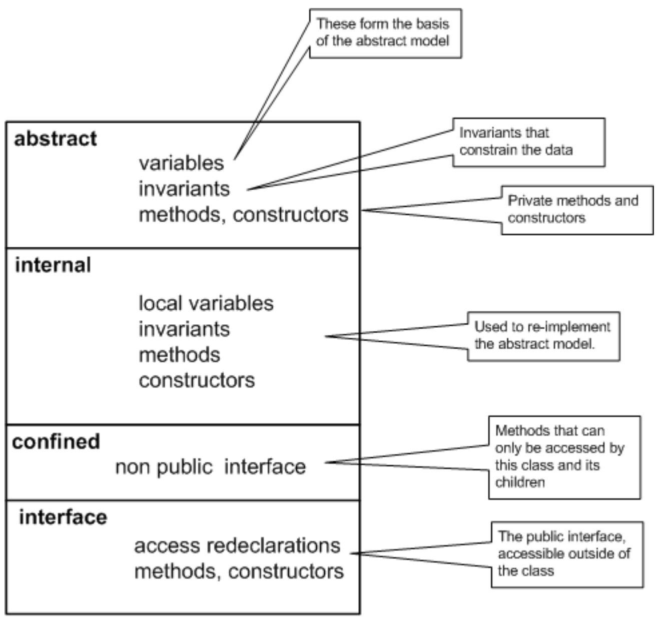
abstract
variables
invariants
methods, constructors
internal
local variables
invariants
methods
constructors
confined
non public interface
interface
access redeclarations
methods, constructors
These form the basis of the abstract model
Invariants that constrain the data
Private methods and constructors
Used to re-implement the abstract model.
Methods that can only be accessed by this class and its children
The public interface, accessible outside of the class

Figure 2: *Perfect Developer* Class

variables within the *CreditCard* example, it would be possible to have *availableFunds* as a secondary variable and this would be expressed within *Perfect* as:

```
function availableFunds : nat
^=
  balance + limit ;

Where ^= means 'is defined as'.
```

## internal

The internal section is available to allow for the refinement of the abstract model and although none exist within the *CreditCard* example an illustration is given below.

```
class Digits ^=
  abstract
    var
      numbers : set of nat;

  internal
    var
        even : set of nat,
        odd : set of nat;

      invariant
      even ** odd = set of nat {} ;

    function numbers ^=
        even ++ odd ;

    interface
    ...
    end;
Where ** means 'set intersection ' and ++ means 'set union '.
```

### confined

The confined section is used to define methods that are available only to the class and its descendants and is similar to *C++'s protected* category.

### interface

This section forms the public interface to the class and may contain the following sorts of declaration:

- function;
  Functions return a result and do not modify the abstract data.
- schema;
  Schemas may modify the abstract data depending on the value of their parameters .
- operator ;
  These are similar to a function but use a symbol rather than a name (i.e. they may be used to define equality between two objects of the same class).
- selector;
  A selector is a method that allows direct read/write access to an item of abstract data.

- constructor;
  The constructor is a method that returns an instance of the class and initialises all of the class's abstract data. Constructors are introduced with the *build* keyword.

### Example

Exploiting the description of the aspects of a *Perfect* class we can take the *CreditCard* example of [7] and express it in *Perfect*, this is illustrated in Listing 1 (shown below).

### Verification

The concept of *Perfect Developer* is *verified design by contract* and thus the environment provides for the verification of the specification and will attempt to prove the consistency and completeness of the abstract model. Once the model has been successfully verified, C++ or Java code can be generated for the target system. As part of the verification process the steps undertaken by the prover are available for inspection and a sample from the *CreditCard* example is shown below.

**Proof of obligation:** Class invariant satisfied

**In the context of class:** CreditCard, declared at: CreditCard.pd (1,7)

**Obligation location:** CreditCard.pd (27,7)

**Condition defined at:** CreditCard.pd (8,11)

**To prove:** **self′**.limit **in set of** nat{2000 **as** nat, 5000 **as** nat, 1000 **as** nat}

**Given:** **self**.limit **in set of** nat{2000 **as** nat, 5000 **as** nat, 1000 **as** nat}, 0 $\leq$ ( **self**.balance + ( **self**.limit **as int**)), 0 $\leq$ amount, ( **self**.balance + (amount **as int**)) = **self′**.balance, $\forall$ \$x $\in$ \$attributeNames(CreditCard) • **different**( **self′**.\$x; self′.balance) $\Rightarrow$ **self**.\$x= **self′**.\$x

**Proof:**

*[Take given term]*

*[2.0]* **self**.limit **in set of** nat{2000 **as** nat, 5000 **as** nat, 1000 **as** nat}

$\rightarrow$ *[remove type check cast]*

*[2.1]* **self**.limit **in set of** nat{2000, 5000 **as** nat, 1000 **as** nat}

$\rightarrow$ *[remove type check cast]*

*[2.2]* **self**.limit **in set of** nat{2000, 5000, 1000 **as** nat}

$\rightarrow$ *[remove type check cast]*

*[2.3]* **self**.limit **in set of** nat{2000, 5000, 1000}

*[Take given term]*

*[7.0]* $\forall$ \$x $\in$ \$attributeNames(CreditCard) • **different**( **self′**.\$x; self′.balance) $\Rightarrow$ **self**.\$x= **self′**.\$x

$\rightarrow$ *[expand all members]*

*[7.1]* ( **self**.limit = **self′**.limit) $\wedge$ ($\forall$ \$x $\in$ \$attributeNames( **int**) • **different**( **self′**.balance.\$x; balance) $\Rightarrow$ **self**.balance.\$x= **self′**.balance.\$x)

$\rightarrow$ *[collect non-literal terms]*

```
class CreditCard ^=
abstract
  var
    limit : nat,
    balance : int;
  invariant
    limit in set of nat {1000, 2000, 5000},
    balance + limit >= 0 ;

interface
  function limit ;
  function balance ;
  schema !INIT
    post
      balance! = 0;
  schema !withdraw (amount : nat)
    pre
      amount <= balance + limit
    post
      balance! = balance - amount;
  schema !deposit (amount : nat)
    post
      balance! = balance + amount;
  schema !withdrawAvail (amount! : out nat)
    post
      amount! = balance + limit ,
      balance! = limit;
  build{}
    post
      balance! = 0,
      limit! = 1000;
end;
```

Listing 1: CreditCard example

*[7.2]* $(0 = (-\ \mathbf{self}'.\text{limit} +\ \mathbf{self}.\text{limit})) \wedge (\forall\ \$x \in \$\text{attributeNames}(\ \mathbf{int}) \bullet\ \mathbf{different}(\ \mathbf{self}'.\text{balance}.\$x;\ \text{balance}) \Rightarrow\ \mathbf{self}.\text{balance}.\$x =\ \mathbf{self}'.\text{balance}.\$x)$

→ *[equal to a modified expression]*

*[7.3]* $(0 = (-\ \mathbf{self}'.\text{limit} +\ \mathbf{self}.\text{limit})) \wedge\ \mathbf{true}$

→ *[evaluate literal expression of the form 'bool & bool']*

*[7.4]* $0 = (-\ \mathbf{self}'.\text{limit} +\ \mathbf{self}.\text{limit})$

*[Take goal term]*

*[1.0]* $\mathbf{self}'.\text{limit}\ \ \mathbf{in}\ \ \mathbf{set}\ \ \mathbf{of}\ \text{nat}\{2000\ \ \mathbf{as}\ \text{nat},\ 5000\ \ \mathbf{as}\ \text{nat},\ 1000\ \ \mathbf{as}\ \text{nat}\}$

→ *[from term 7.4, **self'**.limit is equal to **self**.limit]*

*[1.1]* $\mathbf{self}.\text{limit}\ \ \mathbf{in}\ \ \mathbf{set}\ \ \mathbf{of}\ \text{nat}\{2000\ \ \mathbf{as}\ \text{nat},\ 5000\ \ \mathbf{as}\ \text{nat},\ 1000\ \ \mathbf{as}\ \text{nat}\}$

→ *[remove type check cast]*

*[1.2]* $\mathbf{self}.\text{limit}\ \ \mathbf{in}\ \ \mathbf{set}\ \ \mathbf{of}\ \text{nat}\{2000,\ 5000\ \ \mathbf{as}\ \text{nat},\ 1000\ \ \mathbf{as}\ \text{nat}\}$

→ *[remove type check cast]*

*[1.3]* $\mathbf{self}.\text{limit}\ \ \mathbf{in}\ \ \mathbf{set}\ \ \mathbf{of}\ \text{nat}\{2000,\ 5000,\ 1000\ \ \mathbf{as}\ \text{nat}\}$

→ *[remove type check cast]*

*[1.4]* $\mathbf{self}.\text{limit}\ \ \mathbf{in}\ \ \mathbf{set}\ \ \mathbf{of}\ \text{nat}\{2000,\ 5000,\ 1000\}$

→ *[from term 2.3, **self**.limit **in set of** nat{1000, 2000, 5000} is true]*

*[1.5]* **true**

In a commercial application currently being undertaken part of the specification for a web-enabled database system consists of 35,799 lines of *Perfect* (though a more compact translation may be possible later). The subsequent code generation produces 62,224 lines of Java. In addition, *Perfect Developer* generates 9,819 proof obligations which are all proven automatically. Complete proof by *Perfect Developer* of all the generated obligations requires approximately 41/2 hours on a modest (750MHz) laptop. This represents an average of 1.6 sec/proof, with the longest proof taking 16.2 seconds.

## 3 INTEGRATING *PERFECT DEVELOPER* AND OBJECT-Z

The previous section introduced Object-Z and *Perfect Developer* and illustrated how *Perfect Developer* may be used to express and prove the well known *CreditCard* example. This section will examine other features of Object-Z and show how they may be expressed in *Perfect Developer*.

### Instantiation and interaction

#### Objects as attributes

Like OO languages Object-Z allows a class to be used as the type for an attribute within a later class and for the attribute to be accessed from outside of the class (depending on the visibility list). As *Perfect* is an OO language itself the use of classes as types for later attributes is supported. However, *Perfect* allows the directing reading of an attribute that is redefined as a function but imposes information hiding and normally only allows the modification of an abstract variable via a schema.

## Promoting operations

Within Z and Object-Z the concept of promoting operations exists so that specifications can be broken into manageable parts. Within Object-Z operation promotion is indicated by

$$withdraw_1 \mathrel{\widehat=} c_1.withdraw$$

Within *Perfect* this is achieved via a schema invoking the lower level schema, e.g.

```
schema !withdraw1 (amount : nat)
  post
    c1!withdraw(amount);
```

## Conjunction operator

The conjunction operator within Object-Z is implemented with *Perfect* via the post condition of a schema, e.g.

```
schema !transfer (amount : nat)
  post
    c1!withdraw(amount),
    c1!deposit(amount);
```

And like Object-Z, *Perfect* assumes that the operations occur in parallel.

## Choice operator

The choice operator in Object-Z makes a non deterministic choice between the applicable operations that are valid, e.g.

$$withdrawEither \mathrel{\widehat=} c_1.withdraw [] c_2.withdraw$$

*Perfect* provides deterministic guarded choice using the construct:

([guard1]: postcondition1, [guard2]: postcondition2, ...)

This has the semantics "if guard1 then postcondition 1 else if guard2 then postcondition2 ...".

The choice can be made nondeterministic by inserting the keyword **opaque** after the opening bracket. By making each of the guards **true**, we can translate *withdrawEither* like this:

```
schema !withdrawEither (amount : nat)
  post
    (opaque [true]: c1!.withdraw , [true]: c2!.withdraw);
```

## Inter-object communication

The parallel operator of Obeject-Z provides for inter-object communications, e.g.

$$transferAvail \mathrel{\widehat=} c_1.withdrawAvail \parallel c_2.deposit$$

allows for the results of the 1st operation to be used as input to the 2nd operation. This can be expressed in *Perfect* like this:

```
schema !transferAvail
  post
  (
    var x : nat;
    c1!withdrawAvail(x!) then
    c2!deposit(x)
  );
```

## Inheritance

*Perfect Developer* supports object inheritance and allows for operations to be *virtual* and then defined later (via the *deferred* keyword) and for operations to be redefined in later classes. e.g.

```
class Rule ^=                   class MinimiseRule ^= inherits Rule
abstract                        abstract
  var                             var
    id : Name ,                     unknown : nat;
    firmness : real ;
interface                       interface

// constructor
  build{}                         build {}
    post                            inherits Rule {}
      id! = DEFAULT_NAME,           post
      firmness! = 0.0;                unknown! = 1 ;

  function rating : real          redefine function rating : real
    ^= 0.0 ;                        ^= 1.0 ;
end;                            end;
```

## Polymorphism

Polymorphism allows some relaxation of the typing of an object by allowing it to be in one of a number of classes. Within Object-Z a polymorphic declaration of an object is

$$cards : \mathbb{P} \downarrow CreditCard$$

In *Perfect* the declaration is

```
var cards : from CreditCard
```

*Perfect* support late and dynamic binding and so the operation invoked will depend on the type of the attribute at run time and the parameters associated with the call.

## Class union

Class union allows polymorphism between classes that do not fit neatly fit within an inheritance structure e.g.

$$aircraft == aeroplane \cup helicopter \cup airship$$

This is supported within *Perfect* via its union operator e.g.

```
    class aircraft ^= aeroplane || helicopter || airship ;
```

## Visibility List

*Perfect* does not allow the hiding of the visibility of the inherited public methods as it would break the subtyping and prevent an object of a derived class being substituted for an object of the base class.

## Refinement

Having written a formal specification it is necessary to refine the specification until it can be implemented via a readily verifiable step. With *Perfect Developer* refinement is an integral part of developing, checking and verifying the specification. The *Perfect* language allows the use of the question mark character ("?") to indicate something that has yet to be decided upon. The *check* and *verify* functions of the environment will process the specification and ignore those that have not yet been specified. The *Perfect* language also supports data refinement and procedural refinement.

### Data refinement

Data refinement is achieved via the internal section of the *Perfect* class definition and an example was given earlier in the discussion of a *Perfect* class.

### Procedural refinement

Procedural refinement allows the designer to guide the code generation engine of *Perfect Developer* for example a function to square a value could be expressed using the exponentiation operator (see below).

```
function square (y : int) : int
   ^= y^2 ;
```

However, this gives *Perfect* free rein to implement the function as it wishes, yet it may be known that the target compiler has poor support for exponentiation and a direct multiplication would be more efficient, thus by using the via command the implementation can be directed. e.g.

```
function square (y : int) : int
^= y^2
   via
      value y * y
   end;
```

## 4 DISCUSSION

The information presented in this paper has illustrated how an Object-Z specification may be implemented within *Perfect*, proven and then used to automatically generate code in either Java or C++ and this provides Object-Z developers with an easy route to implementation. Additional motivations for defining a mechanism for implementing Object-Z in *Perfect* was to enable the use of Object-Z and *Perfect Developer* in the design of a potentially critical system and enable the use of software engineers without a strong mathematical background.

The integration described within this paper raises a number of points:

- A two way process;
  The translation between Object-Z and *Perfect* is potentially a two way process and may be used to translate a *Perfect* model into Object-Z. This has the benefit of presenting *Perfect* specifications in the more widely accepted Object-Z format.

- Relationship with other OO technologies?
  The mapping of UML models to Object-Z has been presented in [8] and the

use of OCL with Object-Z has been presented in [9] and so a route from the more common UML description to Object-Z is available. Whereas *Perfect* is still a niche tool with only a limited adoption, it does however, provide support for the importation of UML models described by XMI.

- Teaching formal methods
  The OO nature of Object-Z fits more readily into the OO design paradigm that is currently taught and coupled with the OO nature of *Perfect Developer* they provide a powerful combination to introduce formal methods to students. Some teaching notes have been produced [10]. In addition an academic version of *Perfect Developer* is available free of charge from Escher technologies.

- Will *Perfect* replace Object-Z;
  Although *Perfect Developer* has many features that aid in the development of verifiable applications the application domains that require critical software are normally very conservative and so it is unlikely that it will ever replace Object-Z.

### Lessons Learned

We have taken a practical approach to translating Object-Z to *Perfect* and perhaps strangely for a paper on formal methods have not formally defined the translation but illustrated it with examples. The translation has helped us to implement our Object-Z specification and produce code. It has also illustrated that *Perfect* provides a mechanism for checking the Object-Z specification and thus the translation can be seen as a two process.

## REFERENCES

[1] The OMG, www.omg.org. *Unified Modelling language*, add the current version edition, add year.

[2] J. Warmer & A. Kleppe. *The Object Constraint language*. Addison-Wesley, Menlo Park, California USA, 1999. ISBN 0-201-37940-6.

[3] Graeme Smith. *The Object-Z Specification Language*. Kluwer Academic, 101, Philip Drive, Assinippi Park, Norwell, Ma , 02061. USA, 2000. ISBN 0-7923-8684-1.

[4] J.M. Spivey. *The Z Notation: A reference Manual*. Prentice Hall, 2nd edition, 1998.

[5] Phillip J. Brooke Richard Paige. Integrating bon and object-z. *Journal of Object Technology*, Vol 3, no. 3:121 – 141, March - April 2004.

[6] *Perfect Developer Language Reference Manual.* Escher Technologies, 3.0 edition, December 2004. Available from www.eschertech.com (accessed February 2005).

[7] Roger Duke & Gordon Rose. *Formal Object-Oriented Specification Using Object-Z.* Macmillan Press, Houndmills, Basingstoke, Hampshire. RG21 6XS. UK, 2000. ISBN 0-333-80123-7.

[8] Soon-Kyeong Kim and David A. Carrington. A formal mapping between uml models and object-z specifications. In *ZB '00: Proceedings of the First International Conference of B and Z Users on Formal Specification and Development in Z and B*, pages 2–21. Springer-Verlag, 2000.

[9] Krysia Broda David Roe and Alessandra Russo. Mapping uml models incorporating ocl constraints into object-z. Technical Report 2003/9, Department of Computer Science, Imperial College, London, 2003.

[10] Tony Mullins. Lectures on formal specification and design. World Wide Web, June 2004. available from www.eschertech.com/teaching/tmlecturenotes.zip (accessed March 2005).

## ABOUT THE AUTHORS

**Brian Stevens** is a part time PhD student at Cranfield University and is employed as a researcher, within the aerospace industry, focusing on real-time software architectures for avionic systems. E-Mail b.stevens@cranfield.ac.uk.